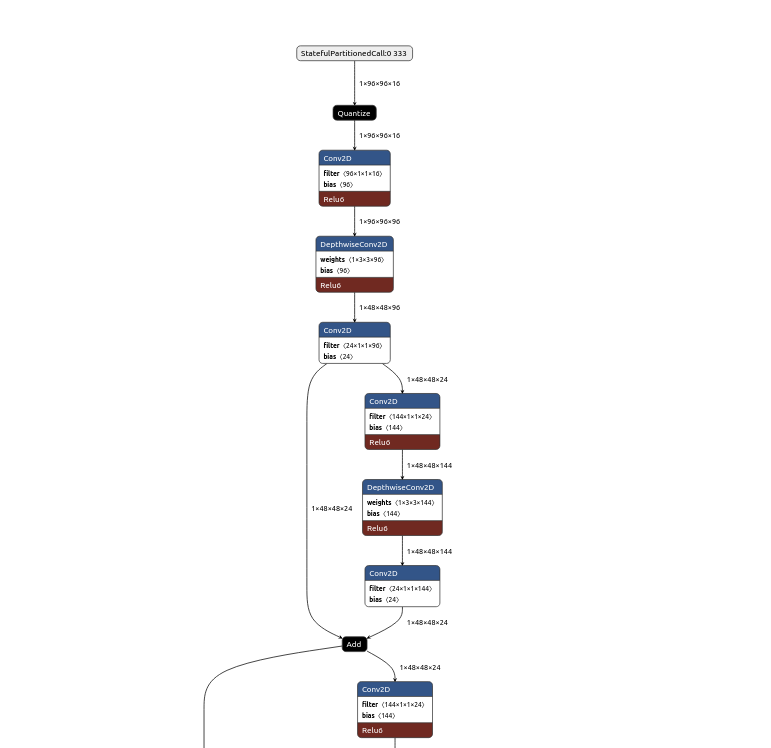
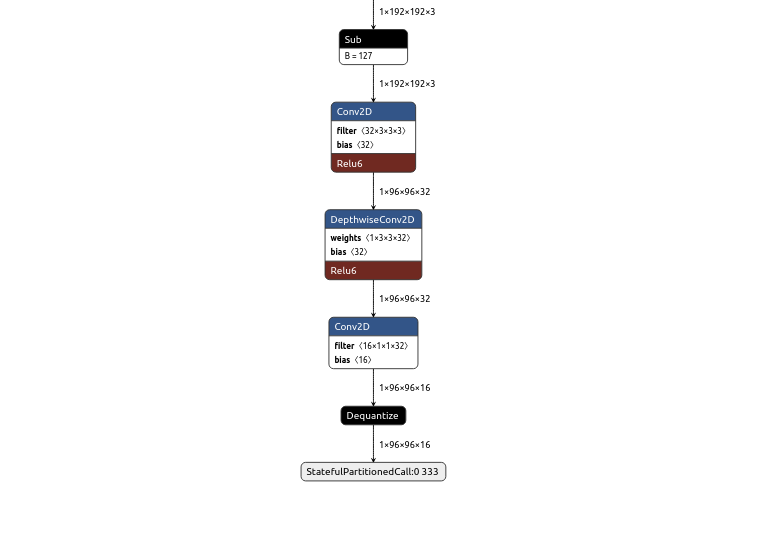
**Assignment #3** – Partitioned, Offloaded DNN Inference

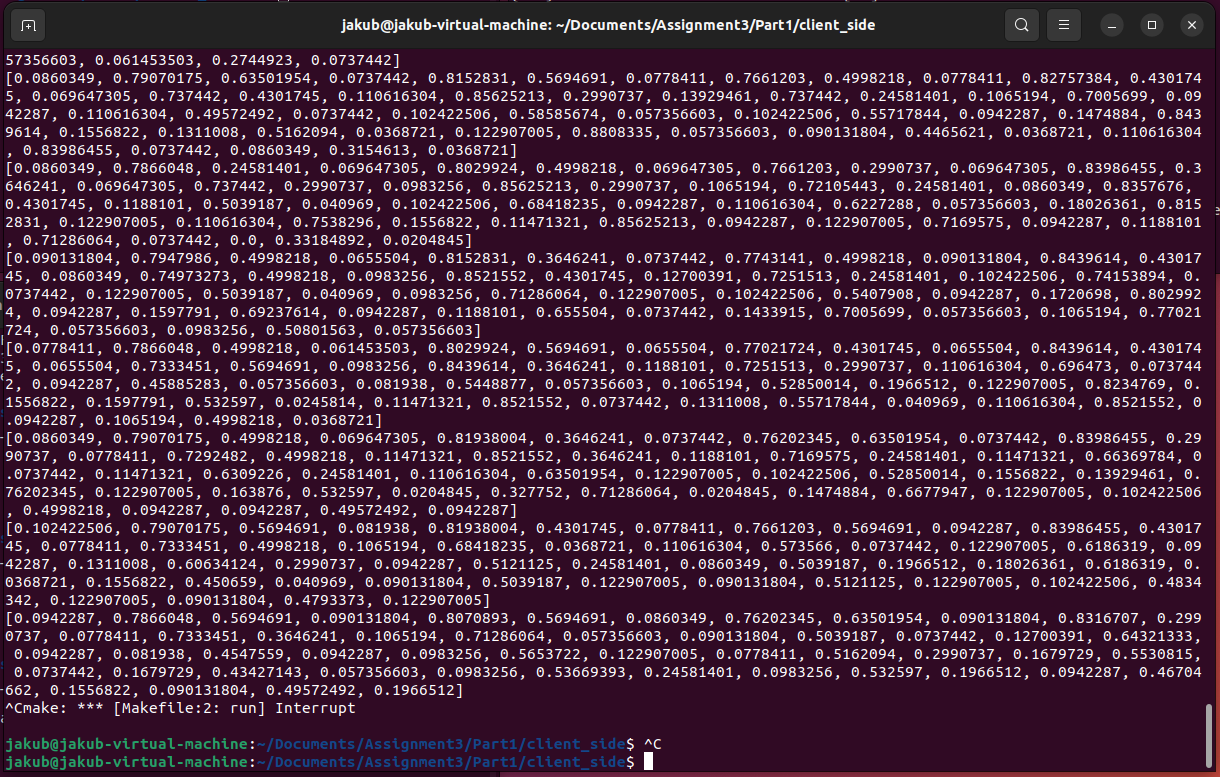
**Part #1** – Partitioned, Offloaded DNN-Based Inference

This part of the assignment was rather simple conceptually, thanks in large part to the Netron app which was very cool to visualize the individual components of the neural net coming together to create the model. In order to do the assignment, I ended up simply using a Python script to convert the model into .json format which I then modified to split it apart. An issue that I ran into while doing this was due to a precision error from within flatc. I addressed this by simply finding all instances of the error (1. / 256 in my case on Node 78), and then simply changing all values to a more precise value. This happened to solve the issue for me. Additionally, I ran into a problem with the data type of the tensors, so I had to convert them into float values and back using the dequantize operator at the end of the local component (left image) and a quantize operator at the beginning of the remote component (right image). After doing this, it was rather simple to get each component to load in the corresponding models and simply use buffers to communicate over the network. I did run into an issue where the output of the OpenCV window solely showed a black screen. I wasn’t able to figure out why it was resulting in a black screen. From the images down below, we see the original display (which can be verified by the path above), and it displays the output along with the 51 float keypoint values that are to be mapped onto the blank canvas. As can be seen in the image above, my **separated** model produces very similar results, however I wasn’t able to figure out how it wasn’t being plotted correctly.

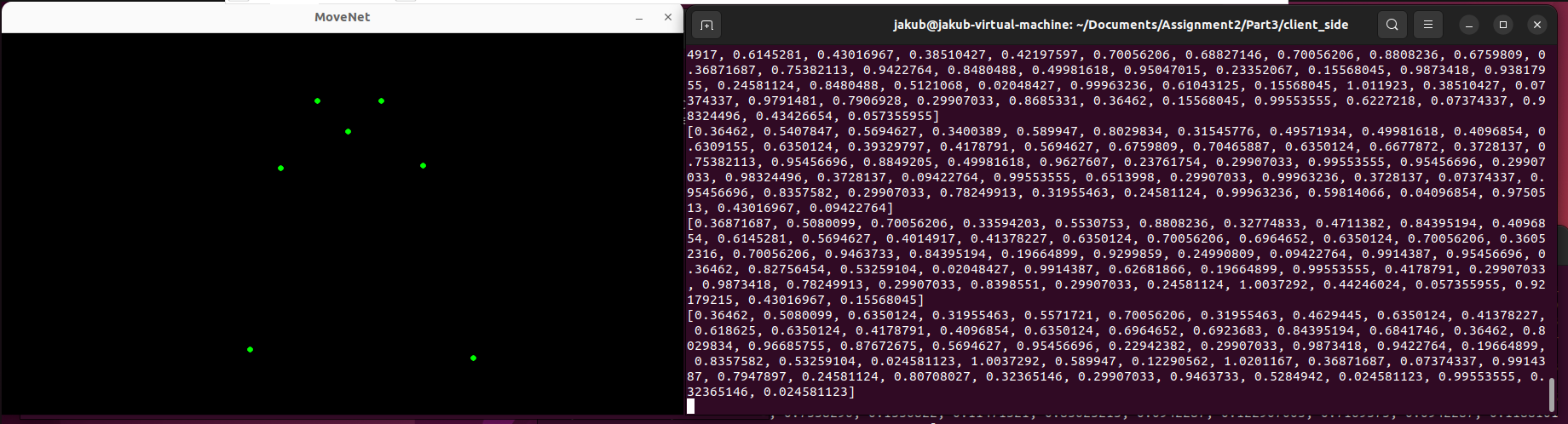
**Local Component/Remote Component**



**Display of Keypoints**



**Original Display**



**Part #2 –** Partitioned, Offloaded DNN-Based Inference in Kernel

Before tackling this assignment, the most that I had dealt with in regards to deep learning models was a quick introduction in Python where I tweaked the parameters and added some layers that I sort of knew what they were doing. However, after this assignment I am much more confident in the use of these layers and how they work more specifically. I had to learn a lot about what each layer is doing specifically in order to be able to hand code it appropriately. However, I was unable to get it hand-coded due to timing issues, so I instead present what I have learned about the assignments, and I will explain how I would’ve went about implementing the corresponding layers that are needed.

More specifically, I began by taking a look at the split tensor flow file, and I saw that I had to implement a couple of operations manually in order to get the code working. The main layers that I had to implement were the cast, quantize, subtraction, multiplication, and convolution layers with ReLU6 activation. The cast, quantize, and operation-based layers were rather simple to come up with since they preserved the shape of the inputted matrix. The Conv2d and DepthwiseConv2d were more difficult, and I wasn’t able to complete this part, due to a lack of time and difficulty. I was unable to get a lot of the code working, and I will instead talk about what I managed to learn about this portion of the assignment.

**cast layer** – This would be a pretty simple for loop through the 1x192x192x3 matrix/array in the Rust file. This would simply be a “for loop” where we use the as command to directly convert the integer values to floats.

**quantize layer** – This layer converts the casted floats into integers using the formula specified in the Netron application.

**subtraction layer** – To my understanding this layer subtracts from every value, although I wasn’t entirely sure.

**multiplication layer** -- To my understanding this layer subtracts from every value, although I wasn’t entirely sure.

**subtraction layer** – This is the same as the subtraction layer from above.

**Conv2d layer** –This layer convolves a filter with while applying a bias to the image. The result is then passed through a ReLU6 activation function.

**DepthwiseConv2d + Relu6** **layer** –This layer is rather complicated in that splits the input into different channels and then convolves with an individual kernel. It then concatenates all of the results along the channels axis. Unlike the previous layer, it doesn’t mix information across different input channels.

**Conv2d layer** –This layer simply convolves the resulting tensor with a set of filters producing thirty-two separate layers each of 96x96 sizes. This time, it is not passed through a ReLU6 activation function but instead fed into the dequantize operation.

**Dequantize** – This layer converts the integers back into floats to be fed into the remote server. This was an important step due to the fact that the outside library does not support using integers as the output. This resulted in an error that I was stuck on for quite a while, but I managed to switch by converting and dealing with everything in floats in the main code.

Overall, given more time, I would’ve been able to implement the assignment quite well, apart from the convolution layers which would definitely require some more understanding.

